ECE 552 / CPS 550
Advanced Computer Architecture I

Lecture 9
Instruction-Level Parallelism – Part 2

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall12.html
27 September – Homework #2 Due
- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

2 October – Class Discussion
Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. “Optimizing pipelines for power and performance”
3. Palacharla et al. “Complexity-effective superscalar processors”
4. Yeh et al. “Two-level adaptive training branch prediction”

4 October – Midterm Exam
In-Order Issue Pipeline

IF → ID → Issue

GPR's  FPR's

ALU → Mem

Fadd

Fmul

Fdiv

WB
Scoreboard

Busy[FU#]: a bit-vector to indicate functional unit availability where FU = \{Int, Add, Mutl, Div\}

WP[#regs]: a bit-vector to record the registers to which writes are pending
- Bits are set to true by issue logic
- Bits are set to false by writeback stage
- Each functional unit’s pipeline registers must carry ‘dest’ field and a flag to indicate if it’s valid: “the (we, ws) pair”

Issue logic checks instruction (opcode, dest, src1, src2) against scoreboard (busy, wp) to dispatch
- FU available? Busy[FU#]
- RAW? WP[src1] or WP[src2]
- WAR? Cannot arise
- WAW? WP[dest]
Limitations of In-Order Issue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: LD</td>
<td>F2, 34(R2)</td>
<td>1</td>
</tr>
<tr>
<td>2: LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3: MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4: SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5: DIVD</td>
<td>F4, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6: ADDD</td>
<td>F10, F6, F4</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2 1) ............2 3 4 4 3 5 ....5 6 6

In-order restriction keeps instruction 4 from issuing
Out-of-Order Issue

- Issue stage buffer holds multiple instructions waiting to issue
- Decode stage adds next instruction to buffer if there is space and next instruction does not cause a WAR or WAW hazard
- Any instruction in buffer whose RAW hazards are satisfied can issue
- When instruction commits, a new instruction can issue
Limitations of Out-of-Order Issue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: LD</td>
<td>F2, 34(R2)</td>
<td>1</td>
</tr>
<tr>
<td>2: LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3: MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4: SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5: DIVD</td>
<td>F4, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6: ADDD</td>
<td>F10, F6, F4</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2 1) ...........2 3 4 4 3 5 ....5 6 6
Out-of-order: 1 (2 1) 4 4 ......2 3...... 3 5 ....5 6 6

Out-of-order execution has no gain.
Why did we not issue instruction 5?
Instructions In-Flight

What features of an ISA limit the number of instructions in the pipeline? **Number of registers**

What features of a program limit the number of instructions in the pipeline? **Control transfers**

Out-of-order issue does not address these other limitations.
Mitigating Limited Register Names

Floating point pipelines often cannot be filled with small number of registers
- IBM 360 had only 4 floating-point registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility?
- In 1967, Robert Tomasulo’s solution was dynamic register renaming.
ILP via Renaming

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: LD</td>
<td>F2, 34(R2)</td>
<td>1</td>
</tr>
<tr>
<td>2: LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3: MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4: SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5: DIVD</td>
<td>F4, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6: ADDD</td>
<td>F10, F6, F4</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2 1) ………….2 3 4 4 3 5 ….5 6 6
Out-of-order: 1 (2 1) 4 4 5 ….2 (3, 5) 3 6 6

Any anti-dependence can be eliminated by renaming (requires additional storage). Renaming can be done in hardware!
Register Renaming

- Decode stage renames registers and adds instructions to the reorder buffer (ROB)
- ROB tracks in-flight instructions in program order
- ROB renames registers to eliminate WAR or WAW hazards
- ROB instructions with resolved RAW hazards can issue (source operands are ready)
- This is called “out-of-order” or “dataflow” execution
Reorder Buffer (ROB)

Instruction slot is candidate for execution when...
- Instruction is valid (“use” bit is set)
- Instruction is not already executing (“exec” bit is clear)
- Operands are available (“p1” and “p2” are set for “src1” and “src2”)
Renaming Registers and the ROB

1. Insert instruction into ROB (after decoding it)
   i. ROB entry is used, use \( \Leftarrow 1 \)
   ii. Instruction is not yet executing, exec \( \Leftarrow 1 \)
   iii. Specify operation in ROB entry

2. Update renaming table
   i. Identify instruction’s destination register (e.g., F1)
   ii. Look up register (e.g., F1) in renaming table
   iii. Insert pointer to instruction’s ROB entry

3. When instruction executes, exec \( \Leftarrow 1 \)

4. When instruction writes-back, replace pointer to ROB with produced value
Example

When are names in sources replaced by data? When a functional unit produces data.

When can a name be re-used? When an instruction completes.
Renaming Registers and the ROB

1. Insert instruction into ROB (after decoding it)
   i. ROB entry is used, use $\leftarrow 1$
   ii. Instruction is not yet executing, exec $\leftarrow 1$
   iii. Specify operation in ROB entry

2. Update renaming table
   i. Identify instruction’s destination register (e.g., F1)
   ii. Look up register (e.g., F1) in renaming table
   iii. Insert pointer to instruction’s ROB entry

3. When instruction executes, exec $\leftarrow 1$

4. When instruction writes-back, replace pointer to ROB with produced value
Register Renaming

- Decode stage allocates instruction template (i.e., tag t) and stores tag in register file.
- When instruction completes, tag is de-allocated.
Allocating/Deallocating Templates

- Reorder buffer is managed **circularly**.
- Field “exec” is set when instruction begins execution.
- Field “use” is cleared when instruction completes
- Ptr2 increments when “use” bit is cleared.

<table>
<thead>
<tr>
<th>Ins#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reorder buffer

ptr₂ → next to deallocate

ptr₁ → next available

- **ptr₂** next to deallocate
- **ptr₁** next available

- **t₁**
- **t₂**
- **.**
- **.**
- **.**
- **.**
Reservation Stations

IBM 360/91 distributes instruction templates (ROB) by functional units. Also known as reservation stations.

Common bus ensures that data is made available immediately to all the instructions waiting for it.

store buffers (to memory)

< t, result >

Floating-point Register File & Renaming Table

load buffers (from memory)

Adder

Mult

Instructions
Effectiveness

History
- Renaming/out-of-order execution first introduction in 360/91 in 1969
- However, implementation did not re-appear until mid-90s
- Why?

Limitations
- Effective on a very small class of problems
- Memory latency was a much bigger problem in the 1960s
- Problem-1: Exceptions were not precise
- Problem-2: Control transfers
Precise Interrupts

Definition
- It must appear as if an interrupt is taken between two instructions
- Consider instructions k, k+1
- Effect of all instructions up to and including k is totally complete
- No effect of any instruction after k has taken place

Interrupt Handler
- Aborts program or restarts at instruction k+1
Out-of-order Completion

- Precise interrupts are difficult to implement at high performance
- Want to start execution of later instructions before exception checks are finished on earlier instructions

\[
\begin{align*}
I_1 & : \text{DIVD} \quad f_6, \quad f_6, \quad f_4 \\
I_2 & : \text{LD} \quad f_2, \quad 45(r3) \\
I_3 & : \text{MULTD} \quad f_0, \quad f_2, \quad f_4 \\
I_4 & : \text{DIVD} \quad f_8, \quad f_6, \quad f_2 \\
I_5 & : \text{SUBD} \quad f_{10}, \quad f_0, \quad f_6 \\
I_6 & : \text{ADDD} \quad f_6, \quad f_8, \quad f_2
\end{align*}
\]

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6
interrupts

restore f2

restore f10
Exception Handling (in-order)

- Hold exception flags in pipeline until commit point
- Exceptions earlier in program order override those later in program order
- Inject external interrupts, which over-ride others, at commit point
- If exception at commit: (1) update Cause and EPC registers, (2) kill all stages, (3) inject handler PC into fetch stage
Phases of Instruction Execution

Fetch: Instruction bits retrieved from cache.

Decode: Instructions placed in appropriate issue (aka “dispatch”) buffer

Execute: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.

Commit: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).
Exception Handling (out-of-order)

In-Order Commit for Precise Exceptions
- Instructions fetched, decoded into reorder buffer (ROB) in-order
- Instructions executed, completed out-of-order
- Instructions committed in-order
- Instruction commit writes to architectural state (e.g., register file, memory)

Need temporary storage for results before commit
Supporting Precise Exceptions

- Add <pd, dest, data, cause> fields to instruction template
  - pd (1 if result ready), dest (target register), data (result computed)
  - cause (reason for interrupt/exception)

- Commit instructions to register file and memory in-order
- On exception, clear re-order buffer (reset ptr-1 = ptr-2)
- Store instructions must commit before modifying memory
Renaming and Rollbacks

Renaming table is a cache, speeds up register name look-up. Table is cleared after each exception. When else are valid bits cleared? Control transfers.
Control Transfer Penalty

Modern processors may have >10 pipeline stages between next PC calculation and branch resolution.

How much work is lost if pipeline does not follow correct instruction flow?

\[\text{Loop Length} \times \text{Pipeline Width}\]
Branches and Jumps

Each instruction fetch depends on 1-2 pieces of information from preceding instruction:
1. Is preceding instruction a branch?
2. If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>after decode</td>
<td>after decode</td>
</tr>
<tr>
<td>JR</td>
<td>after decode</td>
<td>after fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>after fetch*</td>
<td>after decode</td>
</tr>
</tbody>
</table>

*assuming zero? detect when register read
Reducing Control Flow Penalty

Software Solutions
1. Eliminate branches -- loop unrolling increases run length before branch
2. Reduce resolution time -- instruction scheduling moves instruction that produces condition earlier

Hardware Solutions
1. Find other work -- delay slots and software cooperation
2. Speculate -- predict branch result and execute instructions beyond branch
Acknowledgements

These slides contain material developed and copyright by
- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)